CHAPTER 8

Data Analysis Case Studies

In the last chapter, we looked at the various Python-based visualization libraries and how the functions from these libraries can be used to plot different graphs. Now, we aim to understand the practical applications of the concepts we have discussed so far with the help of case studies. We examine the following three datasets:

- Analysis of unstructured data: Using data from a web page providing information about the top 50 highest-grossing movies in France during the year 2018
- Air quality analysis: Data from an air quality monitoring station at New Delhi (India), providing the daily levels for four pollutants – sulfur dioxide (SO_2), oxides of nitrogen as nitrogen dioxide (NO_2), ozone, and fine particulate matter ($PM_{2.5}$)
- COVID-19 trend analysis: Dataset capturing the number of cases and deaths for various countries across the world daily for the first six months in the year 2020

Technical requirements

External files

For the first case study, you need to refer to the following Wikipedia URL (data is taken directly from the web page):

https://en.wikipedia.org/wiki/List_of_2018_box_office_number-one_films_in_France

G. Rajagopalan, *A Python Data Analyst's Toolkit*, https://doi.org/10.1007/978-1-4842-6399-0_8

For the second case study, download a CSV file from the following link:

https://github.com/DataRepo2019/Data-files/blob/master/NSIT%20Dwarka.csv

For the third case study, download an Excel file from the following link: https://github.com/DataRepo2019/Data-files/blob/master/COVID-19-geographic-disbtribution-worldwide-2020-06-29.xlsx

Libraries

In addition to the modules and libraries we used in the previous chapters (including Pandas, NumPy, Matplotlib, and Seaborn), we use the *requests* module in this chapter to make HTTP requests to websites.

To use the functions contained in this module, import this module in your Jupyter notebook using the following line:

```
import requests
```

If the *requests* modules is not installed, you can install it using the following command on the Anaconda Prompt.

```
pip install requests
```

Methodology

We will be using the following methodology for each of the case studies:

1. Open a new Jupyter notebook, and perform the following steps:
    - Import the libraries and data necessary for your analysis
    - Read the dataset and examine the first five rows (using the *head* method)
    - Get information about the data type of each column and the number of non-null values in each column (using the *info* method) and the dimensions of the dataset (using the *shape* attribute)
    - Get summary statistics for each column (using the *describe* method) and obtain the values of the count, min, max, standard deviation, and percentiles

2. Data wrangling
    - Check if the data types of the columns are correctly identified (using the *info* or *dtype* method). If not, change the data types, using the *astype* method
    - Rename the columns if necessary, using the *rename* method
    - Drop any unnecessary or redundant columns or rows, using the *drop* method
    - Make the data tidy, if needed, by restructuring it using the *melt* or *stack* method
    - Remove any extraneous data (blank values, special characters, etc.) that does not add any value, using the *replace* method
    - Check for the presence of null values, using the *isna* method, and drop or fill the null values using the *dropna* or *fillna* method
    - Add a column if it adds value to your analysis
    - Aggregate the data if the data is in a disaggregated format, using the groupby method
3. Visualize the data using univariate, bivariate, and multivariate plots
4. Summarize your insights, including observations and recommendations, based on your analysis

Case study 8-1: Highest grossing movies in France – analyzing unstructured data

In this case study, the data is read from an HTML page instead of a conventional CSV file.

The URL that we are going to use is the following: `https://en.wikipedia.org/wiki/List_of_2018_box_office_number-one_films_in_France`

This web page has a table that displays data about the top 50 films in France by revenue, in the year 2018. We import this data in Pandas using methods from the Requests library. Requests is a Python library used for making HTTP requests. It helps with extracting HTML from web pages and interfacing with APIs.

Questions that we want to answer through our analysis:

1. Identify the top five films by revenue
2. What is the percentage share (revenue) of each of the top ten movies?
3. How did the monthly average revenue change during the year?

Step 1: Importing the data and examining the characteristics of the dataset

First, import the libraries and use the necessary functions to retrieve the data.

CODE:

```
#importing the libraries
import requests
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
#Importing the data from the webpage into a DataFrame
url='https://en.wikipedia.org/wiki/List_of_2018_box_office_number-one_
films_in_France'
req=requests.get(url)
data=pd.read_html(req.text)
df=data[0]
```

We import all the libraries and store the URL in a variable. Then we make an HTTP request to this URL using the get method to retrieve information from this web page. The text attribute of the requests object contains the HTML data, which is passed to the *pd.read_html* function. This function returns a list of DataFrame objects containing the various tables on the web page. Since there is only one table on the web page, the DataFrame (df) contains only one table.

Examining the first few records:

CODE:

```
df.head()
```

Output:

	#	Date	Film	Gross	Notes
0	1	January 7, 2018	Star Wars: The Last Jedi	US$6,557,062	[1]
1	2	January 14, 2018	Jumanji: Welcome to the Jungle	US$2,127,871	[2]
2	3	January 21, 2018	Brillantissime	US$2,006,033	[3]
3	4	January 28, 2018	The Post	US$2,771,269	[4]
4	5	February 4, 2018	Les Tuche 3	US$16,604,101	[5]

Obtaining the data types and missing values:

CODE:

```
df.info()
```

Output:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 50 entries, 0 to 49
Data columns (total 5 columns):
#        50 non-null int64
Date     50 non-null object
Film     50 non-null object
Gross    50 non-null object
Notes    50 non-null object
dtypes: int64(1), object(4)
memory usage: 2.0+ KB
```

As we can see, the data types of the columns are not in the format we need. The "Gross" column represents the gross revenue, which is a numeric column. This column, however, has been assigned an object data type because it contains numeric as well as non-numeric data (characters like ",", "$" symbol, and letters like "U" and "S"). In the next step, we deal with problems such as these.

Step 2: Data wrangling

In this step, we will:

1. Remove unnecessary characters
2. Change data types

3. Remove columns that are not needed
4. Create a new column from an existing column

Let us remove the unwanted strings from the "Gross" column, retaining only the numeric values:

CODE:

```
#removing unnecessary characters from the Gross column
df['Gross']=df['Gross'].str.replace(r"US\$","").str.replace(r",","")
```

In the preceding statement, we use a series of chained replace methods and the principle of regular expressions to replace the non-numeric characters. The first replace method removes "US$" and the second replace method removes the commas. Replacing a character with an empty string ("") is equivalent to removing the character.

Now, let us use the *astype* method to typecast or change the data type of this column to *int64* so that this column can be used for computations and visualizations:

CODE:

```
#changing the data type of the Gross column to make the column numeric
df['Gross']=df['Gross'].astype('int64')
```

To check whether these changes have been reflected, we examine the first few records of this column and verify the data type:

CODE:

```
df['Gross'].head(5)
```

Output:

```
0     6557062
1     2127871
2     2006033
3     2771269
4    16604101
Name: Gross, dtype: int64
```

As we can see from the output, the data type of this column has been changed, and the values do not contain strings any longer.

We also need to extract the month part of the date, which we will do by first changing the data type of the "Date" column and then applying the *DatetimeIndex* method to it, as shown in the following.

CODE:

```
#changing the data type of the Date column to extract its components
df['Date']=df['Date'].astype('datetime64')
#creating a new column for the month
df['Month']=pd.DatetimeIndex(df['Date']).month
```

Finally, we remove two unnecessary columns from the DataFrame, using the following statement.

CODE:

```
#dropping the unnecessary columns
df.drop(['#','Notes'],axis=1,inplace=True)
```

Step 3: Visualization

To visualize our data, first we create another DataFrame (df1), which contains a subset of the columns the original DataFrame (df) contains. This DataFrame, df1, contains just two columns – "Film" (the name of the movie) and "Gross" (the gross revenue). Then, we sort the values of the revenue in the descending order. This is shown in the following step.

CODE:

```
df1=df[['Film','Gross']].sort_values(ascending=False,by='Gross')
```

There is an unwanted column ("index") that gets added to this DataFrame that we will remove in the next step.

CODE:

```
df1.drop(['index'],axis=1,inplace=True)
```

Top Five Films:

The first plot we create is a bar graph showing the top five films in terms of revenue: (Figure 8-1).

```
#Plotting the top 5 films by revenue
#setting the figure size
plt.figure(figsize=(10,5))
#creating a bar plot
```

```
ax=sns.barplot(x='Film',y='Gross',data=df1.head(5))
#rotating the x axis labels
ax.set_xticklabels(labels=df1.head()['Film'],rotation=75)
#setting the title
ax.set_title("Top 5 Films per revenue")
#setting the Y-axis labels
ax.set_ylabel("Gross revenue")
#Labelling the bars in the bar graph
for p in ax.patches:
ax.annotate(p.get_height(),(p.get_x()+p.get_width()/2,p.get_height()),
ha='center',va='bottom')
```

Output:

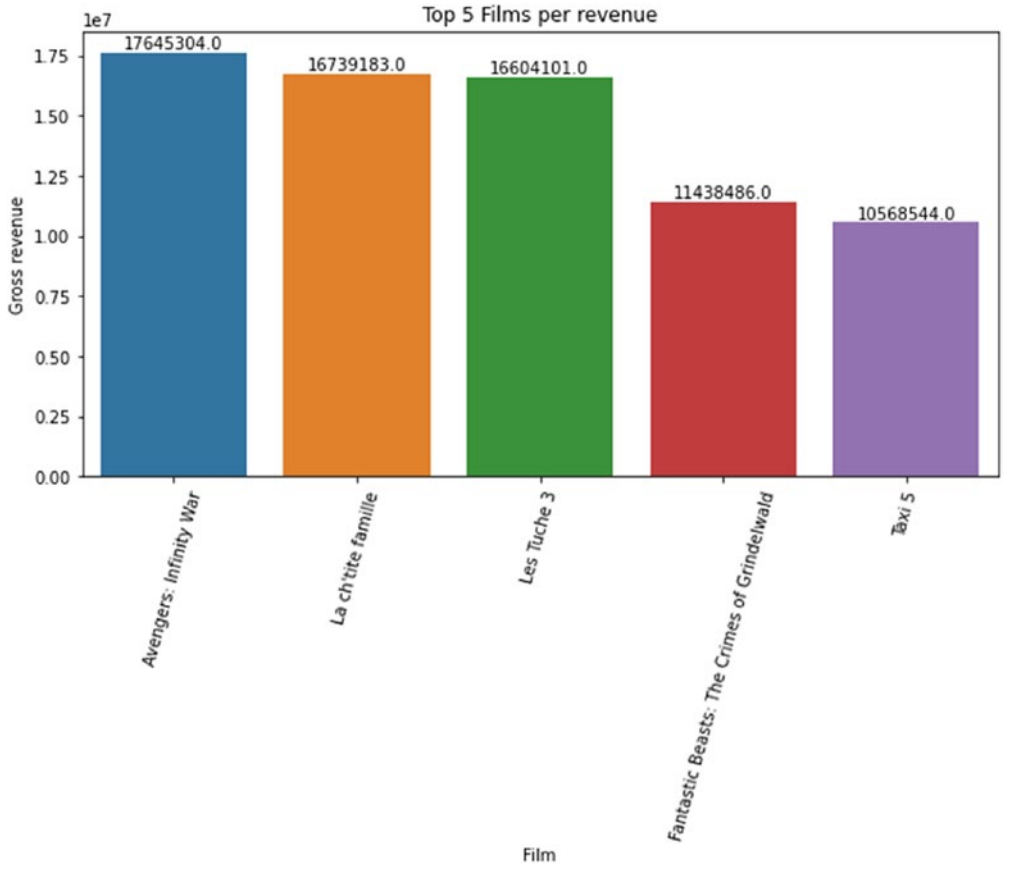

Figure 8-1. *Pie chart showing the individual percentage share of each of the top five films*

To depict the share of the top ten films (by revenue), we create a pie chart (Figure 8-2).

CODE:

```
#Pie chart showing the share of each of the top 10 films by percentage in
the revenue
df1['Gross'].head(10).plot(kind='pie',autopct='%.2f%%',labels=df1['Film'],
figsize=(10,5))
```

Output:

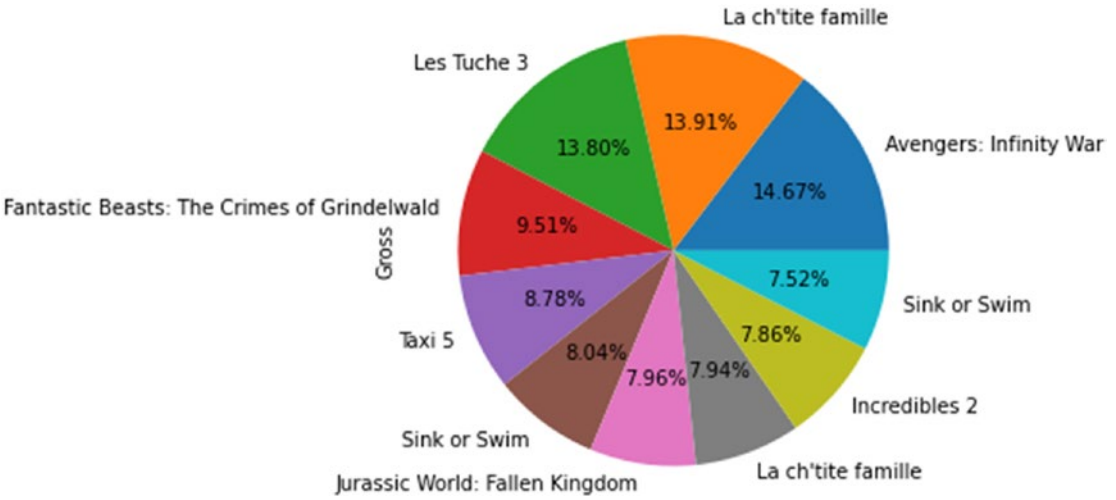

Figure 8-2. *Pie chart depicting the share of top ten movies by revenue*

We first create another DataFrame that aggregates the data for a month by calculating an average for each month (Figure 8-3).

CODE:

```
#Aggregating the revenues by month
df2=df.groupby('Month')['Gross'].mean()
#creating a line plot
df2.plot(kind='line',figsize=(10,5))
```

Output:

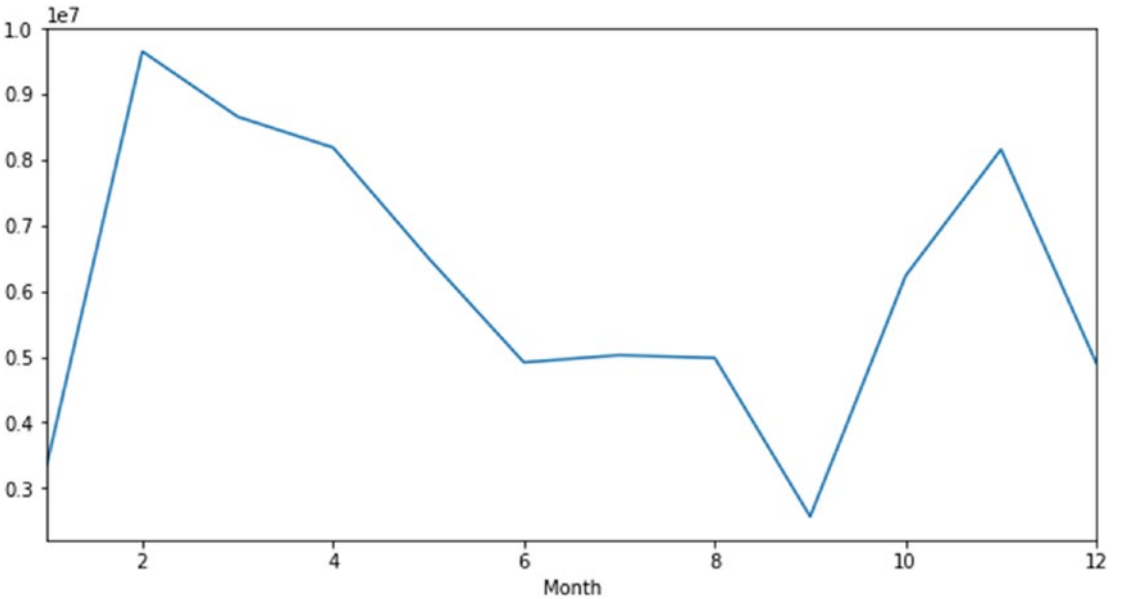

Figure 8-3. *Average box office monthly revenue in 2018 (France)*

Step 4: Drawing inferences based on analysis and visualizations

1. The average monthly revenue shows wide variation, possibly dependent on the month of release of the movies, which may necessitate further analysis across the years.
2. The top three highest-revenue-grossing movies in France in the year 2018 were *Avengers*, *La Ch'tite Famille*, and *Les Tuche 3*.

Case study 8-2: Use of data analysis for air quality management

To monitor the status of ambient air quality, The Central Pollution Control Board (CPCB), India, operates a vast network of monitoring stations spread across the country. Parameters regularly monitored include sulfur dioxide (SO_2), oxides of nitrogen as nitrogen dioxide (NO_2), ozone, and fine particulate matter ($PM_{2.5}$). Based on trends over the years, air quality in the national capital of Delhi has emerged as a matter of public

concern. A stepwise analysis of daily air quality data follows to demonstrate how data analysis could assist in planning interventions as part of air quality management.

Note: The name of the dataset used for this case study is: "NSIT Dwarka.csv". Please refer to the technical description section for details on how to import this dataset.

Questions that we want to answer through our analysis:

1. Yearly averages: Out of the four pollutants - SO_2, NO_2, ozone, and $PM_{2.5}$ - which have yearly average levels that regularly surpass the prescribed annual standards?
2. Daily standards: For the pollutants of concern, on how many days in each year are the daily standards exceeded?
3. Temporal variation: Which are the months where the pollution levels exceed critical levels on most days?

Step 1: Importing the data and examining the characteristics of the dataset

CODE:

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
#aqdata is the name of the DataFrame, short for Air Quality Data.
aqdata=pd.read_csv('NSIT Dwarka.csv')
aqdata.head()
```

Output:

	From Date	To Date	PM2.5	SO2	Ozone	NO2
0	01-01-2014 00:00	02-01-2014 00:00	None	22.7	8.63	5.59
1	02-01-2014 00:00	03-01-2014 00:00	None	8.72	8.43	3.68
2	03-01-2014 00:00	04-01-2014 00:00	None	13.83	9.77	3.83
3	04-01-2014 00:00	05-01-2014 00:00	None	27.64	6.83	9.64
4	05-01-2014 00:00	06-01-2014 00:00	None	37.17	7.34	11.06

Checking for the data types of the columns:

CODE:

```
aqdata.info()
```

Output:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2192 entries, 0 to 2191
Data columns (total 6 columns):
From Date    2191 non-null object
To Date      2191 non-null object
PM2.5        2191 non-null object
SO2          2187 non-null object
Ozone        2187 non-null object
NO2          2190 non-null object
dtypes: object(6)
memory usage: 102.8+ KB
```

Observation: Even though the values for SO_2, NO_2, ozone, and $PM_{2.5}$ are numeric, Pandas reads the data type of these columns as "*object*". To work with these columns (i.e., plot graphs, observe trends, calculate aggregate values), we need to change the data types of these columns. Also, there seem to be some missing entries.

Step 2: Data wrangling

Based on the observations in the previous step, in this step, we will

1. Deal with missing values: We have the option of either dropping the null values or substituting the null values.
2. Change the data types for the columns.

Checking for missing values in the dataset:

CODE:

```
aqdata.isna().sum()
```

Output:

```
From Date    1
To Date      1
PM2.5        1
SO2          5
Ozone        5
NO2          2
dtype: int64
```

There do not seem to be many missing values, but herein lies the catch. When we examined the first few rows using the head statement, we saw that some missing values are represented as *None* in the original dataset. However, these are not being recognized as null values by Pandas. Let us replace the value, *None*, with the value *np.nan* so that Pandas acknowledges these values as null values:

CODE:

```
aqdata=aqdata.replace({'None':np.nan})
```

Now, if we count the number of null values, we see a vastly different picture, indicating a much higher presence of missing values in the dataset.

CODE:

```
aqdata.isna().sum()
```

Output:

```
From Date      1
To Date        1
PM2.5        562
SO2           84
Ozone        106
NO2          105
dtype: int64
```

Let us check the current data types of the columns:

CODE:

```
aqdata.info()
```

Output:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2192 entries, 0 to 2191
Data columns (total 6 columns):
From Date    2191 non-null object
To Date      2191 non-null object
PM2.5        1630 non-null object
SO2          2108 non-null object
Ozone        2086 non-null object
NO2          2087 non-null object
dtypes: object(6)
memory usage: 102.8+ KB
```

We see that the columns containing numeric values are not recognized as numeric columns, and the columns containing dates are also not recognized correctly. Having columns with incorrect data types becomes an impediment for the next step, where we analyze trends and plot graphs; this step requires the data types of the columns to be in a format that is appropriate for Pandas to read.

In the following lines of code, we use the *pd.to_datetime* method to convert the data type of the "From Date" and "To Date" columns to the *datetime* type, which makes it easier to analyze individual components of the date like months and years.

CODE:

```
aqdata['From Date']=pd.to_datetime(aqdata['From Date'], format='%d-%m-%Y %H:%M')
aqdata['To Date']=pd.to_datetime(aqdata['To Date'], format='%d-%m-%Y %H:%M')
aqdata['SO2']=pd.to_numeric(aqdata['SO2'],errors='coerce')
aqdata['NO2']=pd.to_numeric(aqdata['NO2'],errors='coerce')
aqdata['Ozone']=pd.to_numeric(aqdata['Ozone'],errors='coerce')
aqdata['PM2.5']=pd.to_numeric(aqdata['PM2.5'],errors='coerce')
```

Use the *info* method to check whether the data types have been changed.

CODE:

```
aqdata.info()
```

Output:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2192 entries, 0 to 2191
Data columns (total 6 columns):
From Date    2191 non-null datetime64[ns]
To Date      2191 non-null datetime64[ns]
PM2.5        1630 non-null float64
SO2          2108 non-null float64
Ozone        2086 non-null float64
NO2          2087 non-null float64
dtypes: datetime64[ns](2), float64(4)
memory usage: 102.8 KB
```

Since most of our analysis considers yearly data, we create a new column to extract the year, using the *pd.DatetimeIndex* function.

CODE:

```
aqdata['Year'] = pd.DatetimeIndex(aqdata['From Date']).year
```

Now, we create separate DataFrame objects for each year so that we can analyze the data yearly.

CODE:

```
#extracting the data for each year
aq2014=aqdata[aqdata['Year']==2014]
aq2015=aqdata[aqdata['Year']==2015]
aq2016=aqdata[aqdata['Year']==2016]
aq2017=aqdata[aqdata['Year']==2017]
aq2018=aqdata[aqdata['Year']==2018]
aq2019=aqdata[aqdata['Year']==2019]
```

Now, let us have a look at the number of null values in the data for each year:

CODE:

```
#checking the missing values in 2014
aq2014.isna().sum()
```

Output:

```
From Date      0
To Date        0
PM2.5        365
SO2            8
Ozone          8
NO2            8
Year           0
dtype: int64
```

```
#checking the missing values in 2015
aq2015.isna().sum()
```

Output:

```
From Date      0
To Date        0
PM2.5        117
SO2           12
Ozone         29
NO2           37
Year           0
dtype: int64
```

CODE:

```
#checking the missing values in 2016
aq2016.isna().sum()
```

Output:

```
From Date     0
To Date       0
PM2.5        43
SO2          43
Ozone        47
NO2          42
Year          0
dtype: int64
```

```
#checking the missing values in 2017
aq2017.isna().sum()
```

Output:

```
From Date     0
To Date       0
PM2.5        34
SO2          17
Ozone        17
NO2          12
Year          0
dtype: int64
```

CODE:

```
#checking the missing values in 2018
aq2018.isna().sum()
```

Output:

```
From Date    0
To Date      0
PM2.5        2
SO2          2
Ozone        2
NO2          2
Year         0
dtype: int64
```

CODE:

```
#checking the missing values in 2019
aq2019.isna().sum()
```

Output:

```
From Date    0
To Date      0
PM2.5        0
SO2          1
Ozone        2
NO2          3
Year         0
dtype: int64
```

From the analysis of null values for each year, we see that data for the years 2014 and 2015 have the majority of the missing values. Hence, we choose to disregard data from the years 2014 and 2015, and analyze data for 4 years from 2016 to 2019. As per norms set by the Central Pollution Control Board, India, we need at least 104 daily monitored values to arrive at annual averages.

2016, 2017, 2018, and 2019 are the four years for which air quality data would be analyzed. Before moving on to the next step, we drop the missing values for each year from 2016 to 2019 instead of substituting them since we have sufficient data (more than 104 readings) for each of these four years to calculate annual averages, as shown below.

CODE:

```
#dropping the null values for the four years chosen for analysis
aq2016.dropna(inplace=True) # inplace=True makes changes in the original
dataframe
aq2017.dropna(inplace=True)
aq2018.dropna(inplace=True)
aq2019.dropna(inplace=True)
```

Step 3: Data visualization

Part 1 of analysis: Plotting the yearly averages of the pollutants

Based on monitored 24-hourly average ambient air concentrations of $PM_{2.5}$, SO_2, NO_2, and ozone (O_3), yearly averages are plotted to identify parameters for which the prescribed national ambient air quality standards for annual average are exceeded.

First, we calculate the yearly averages for each pollutant ($PM_{2.5}$, SO_2, NO_2, and ozone), as follows:

CODE:

```
#Yearly averages for SO2 in each year
s16avg=round(aq2016['SO2'].mean(),2)
s17avg=round(aq2017['SO2'].mean(),2)
s18avg=round(aq2018['SO2'].mean(),2)
s19avg=round(aq2019['SO2'].mean(),2)
#Yearly averages for PM2.5 in each year
p16avg=round(aq2016['PM2.5'].mean(),2)
p17avg=round(aq2017['PM2.5'].mean(),2)
```

```
p18avg=round(aq2018['PM2.5'].mean(),2)
p19avg=round(aq2019['PM2.5'].mean(),2)
#Yearly averages for NO2 in each year
n16avg=round(aq2016['NO2'].mean(),2)
n17avg=round(aq2017['NO2'].mean(),2)
n18avg=round(aq2018['NO2'].mean(),2)
n19avg=round(aq2019['NO2'].mean(),2)
```

Explanation: The notation for naming variables representing the averages of pollutants is as follows: the first letter of the pollutant, the year, and the abbreviation "avg" for average. For instance, s15avg denotes the average level of SO_2 in the year 2015. We use the *mean* method to calculate the average and the round function to round the average value to two decimal points. We do not consider ozone since yearly standards do not apply to ozone.

Next, we create a DataFrame for each pollutant with two columns each. One of the columns represents the year, and the other column shows the yearly average level for that year.

CODE:

```
#Creating data frames with yearly averages for each pollutant
dfs=pd.DataFrame({'Yearly average':[s16avg,s17avg,s18avg,s19avg]},ind
ex=['2016','2017','2018','2019']) #dfs is for SO2
dfp=pd.DataFrame({'Yearly average':[p16avg,p17avg,p18avg,p19avg]},ind
ex=['2016','2017','2018','2019']) #dfp is for PM2.5
dfn=pd.DataFrame({'Yearly average':[n16avg,n17avg,n18avg,n19avg]},ind
ex=['2016','2017','2018','2019']) #dfn is for NO2
```

Now, we are ready to plot the graphs for the yearly averages of each pollutant (Figure 8-4).

CODE:

```
#Creating a figure with 3 subplots - 1 for each pollutant
fig,(ax1,ax2,ax3)=plt.subplots(1,3)
#Creating a DataFrame the yearly averages for NO2
dfn.plot(kind='bar',figsize=(20,5),ax=ax1)
#Setting the title for the first axes object
ax1.set_title("NO2", fontsize=18)
```

```
#Setting the X-axis label for the NO2 graph
ax1.set_xlabel("Years", fontsize=18)
ax1.legend().set_visible(False)
#Setting the Y-axis label
ax1.set_ylabel("Yearly average", fontsize=18)
#Creating a dashed line to indicate the annual standard
ax1.hlines(40, -.9,15, linestyles="dashed")
#Labelling this dashed line
ax1.annotate('Annual avg. standard for NO2',(-0.5,38))
#labelling the bars
for p in ax1.patches:
    ax1.annotate(p.get_height(),(p.get_x()+p.get_width()/2,p.get_height()),
    color="black", ha="left", va ='bottom',fontsize=12)

#Plotting the yearly averages similarly for PM2.5
dfp.plot(kind='bar',figsize=(20,5),ax=ax2)
ax2.set_title("PM2.5", fontsize=18)
ax2.hlines(40, -.9,15, linestyles="dashed")
ax2.annotate('Annual avg. standard for PM2.5',(-0.5,48))
ax2.legend().set_visible(False)
for p in ax2.patches:
    ax2.annotate(p.get_height(),(p.get_x()+p.get_width()/2,p.get_height()),
    color="black", ha="center", va ='bottom',fontsize=12)

#Plotting the yearly averages similarly for SO2
dfs.plot(kind='bar',figsize=(20,5),ax=ax3)
ax3.hlines(50, -.9,15, linestyles="dashed")
ax3.annotate('Annual avg. standard for SO2',(-0.5,48))
ax3.set_title("SO2", fontsize=18)
ax3.legend().set_visible(False)
for p in ax3.patches:
    ax3.annotate(p.get_height(),(p.get_x()+p.get_width()/2,p.get_height()),
    color="black", ha="center", va ='bottom',fontsize=12)
```

Output:

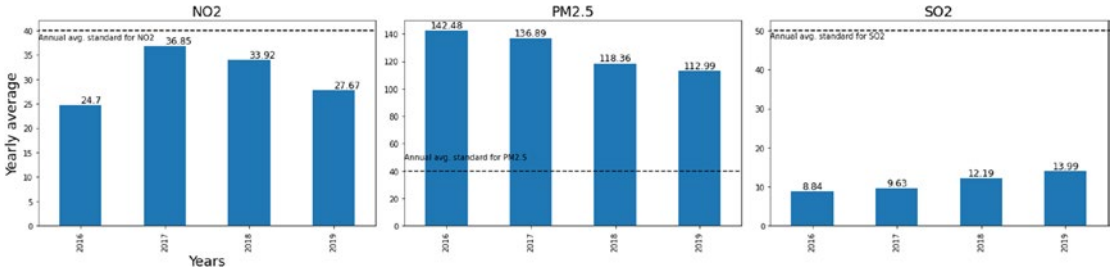

Figure 8-4. *The yearly average level for the pollutants (NO_2, $PM_{2.5}$, and SO_2), vis-à-vis their annual average standard*

Observation: It is evident that standards for annual average are exceeded only for $PM_{2.5}$. For NO_2, the observed values are relatively close to the prescribed standard. For SO_2, the observed values are much less than the annual standard. Therefore, for further analysis, only these two pollutants (NO_2 and $PM_{2.5}$) are considered.

Part 2 of air quality analysis: Plotting the number of days in each year where 24-hourly standards are exceeded for $PM_{2.5}$ and NO_2

While step 1 of the analysis indicates pollutants of concern for air quality management and planning of interventions, in step 2, for each year, we show how various levels of exceedance above standards for 24-hourly values are distributed. In the case of $PM_{2.5}$, we plot the number of days in each year for which observed values fall in the following ranges.

a. 0 to 60 µg/m^3

b. 61 to 120 µg/m^3

c. 121 to 180 µg/m^3

d. > 180 µg/m^3

To plot this data, we need to create DataFrame objects for each year from 2016 to 2019 capturing the number of days with $PM_{2.5}$ levels falling in each of these intervals, as shown in the following:

CODE:

```
#Creating intervals for 2016 with the number of days with PM2.5
concentration falling in that interval
a2=aq2016[(aq2016['PM2.5']<=60)]['PM2.5'].count()
```

```
b2=aq2016[((aq2016['PM2.5']>60) & (aq2016['PM2.5']<=120))]['PM2.5'].count()
c2=aq2016[((aq2016['PM2.5']>120) & (aq2016['PM2.5']<=180))]['PM2.5'].count()
d2=aq2016[(aq2016['PM2.5']>180)]['PM2.5'].count()
dfpb2016=pd.DataFrame({'year':'2016','pm levels':['<60','between 61
and 120','between 121 and 180','greater than 180'],'number of critical
days':[a2,b2,c2,d2]})
#Creating intervals for 2017 with the number of days with PM2.5
concentration falling in each interval
a3=aq2017[(aq2017['PM2.5']<=60)]['PM2.5'].count()
b3=aq2017[((aq2017['PM2.5']>60) & (aq2017['PM2.5']<=120))]['PM2.5'].count()
c3=aq2017[((aq2017['PM2.5']>120) & (aq2017['PM2.5']<=180))]['PM2.5'].count()
d3=aq2017[(aq2017['PM2.5']>180)]['PM2.5'].count()
dfpb2017=pd.DataFrame({'year':'2017','pm levels':['<60','between 61
and 120','between 121 and 180','greater than 180'],'number of critical
days':[a3,b3,c3,d3]})
#Creating intervals for 2018 with the number of days with PM2.5
concentration falling in each interval
a4=aq2018[(aq2018['PM2.5']<=60)]['PM2.5'].count()
b4=aq2018[((aq2018['PM2.5']>60) & (aq2018['PM2.5']<=120))]['PM2.5'].count()
c4=aq2018[((aq2018['PM2.5']>120) & (aq2018['PM2.5']<=180))]['PM2.5'].count()
d4=aq2018[(aq2018['PM2.5']>180)]['PM2.5'].count()
dfpb2018=pd.DataFrame({'year':'2018','pm levels':['<60','between 61
and 120','between 121 and 180','greater than 180'],'number of critical
days':[a4,b4,c4,d4]})
#Creating intervals for 2019 with the number of days with PM2.5
concentration falling in each interval
a5=aq2019[(aq2019['PM2.5']<=60)]['PM2.5'].count()
b5=aq2019[((aq2019['PM2.5']>60) & (aq2019['PM2.5']<=120))]['PM2.5'].count()
c5=aq2019[((aq2019['PM2.5']>120) & (aq2019['PM2.5']<=180))]['PM2.5'].
count()
d5=aq2019[(aq2019['PM2.5']>180)]['PM2.5'].count()
dfpb2019=pd.DataFrame({'year':'2019','pm levels':['<60','between 61
and 120','between 121 and 180','greater than 180'],'number of critical
days':[a5,b5,c5,d5]})
```

Now, we plot a stacked bar chart for each year with these intervals. To do so, we need to create pivot tables as follows:

CODE:

```
dfpivot2019=dfpb2019.pivot(index='year',columns='pm levels',values='number
of critical days')
dfpivot2018=dfpb2018.pivot(index='year',columns='pm levels',values='number
of critical days')
dfpivot2017=dfpb2017.pivot(index='year',columns='pm levels',values='number
of critical days')
dfpivot2016=dfpb2016.pivot(index='year',columns='pm levels',values='number
of critical days')
```

Using these pivot tables, we create stacked bar charts (Figure 8-5) as follows:

CODE:

```
#Creating a figure with 4 sub-plots, one for each year from 2016-19
fig,(ax1,ax2,ax3,ax4)=plt.subplots(1,4)
fig.suptitle("Number of days per year in each interval")
cmp=plt.cm.get_cmap('RdBu')
#Plotting stacked horizontal bar charts for each year to represent
intervals of PM2.5 levels
dfpivot2019.loc[:,['<60','between 61 and 120','between 121 and 180',
'greater than 180']].plot.barh(stacked=True, cmap=cmp,figsize=(15,5),ax=ax1)
dfpivot2018.loc[:,['<60','between 61 and 120','between 121 and
180','greater than 180']].plot.barh(stacked=True, cmap=cmp,
figsize=(15,5),ax=ax2)
dfpivot2017.loc[:,['<60','between 61 and 120','between 121 and
180','greater than 180']].plot.barh(stacked=True, cmap=cmp,
figsize=(15,5),ax=ax3)
dfpivot2016.loc[:,['<60','between 61 and 120','between 121 and
180','greater than 180']].plot.barh(stacked=True, cmap=cmp,
figsize=(15,5),ax=ax4)
#Setting the properties - legend, yaxis and title
ax1.legend().set_visible(False)
ax2.legend().set_visible(False)
ax3.legend().set_visible(False)
```

```
ax4.legend(loc='center left',bbox_to_anchor=(1,0.5))
ax1.get_yaxis().set_visible(False)
ax2.get_yaxis().set_visible(False)
ax3.get_yaxis().set_visible(False)
ax4.get_yaxis().set_visible(False)
ax1.set_title('2019')
ax2.set_title('2018')
ax3.set_title('2017')
ax4.set_title('2016')
```

Output:

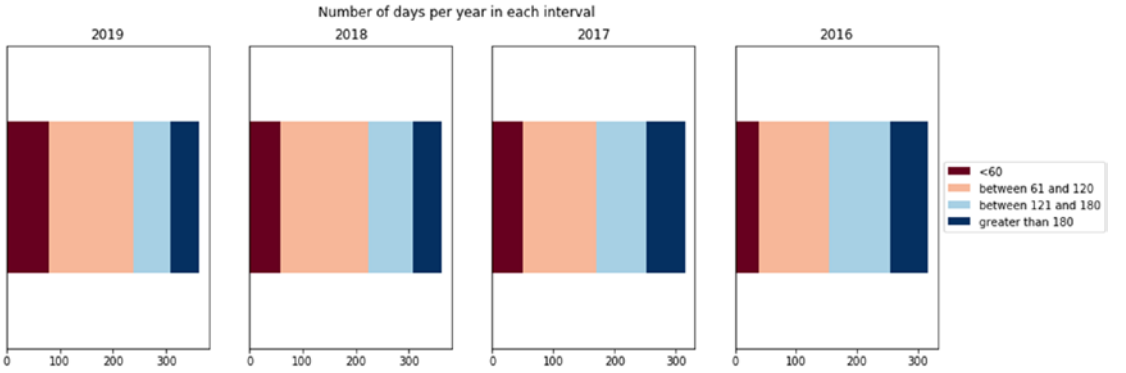

Figure 8-5. *Number of days per year in each interval level for $PM_{2.5}$*

Observation:

It is seen that $PM_{2.5}$ values above 180 µg/m^3 are observed every year, and therefore, to start with, restrictions on major polluting activities, including traffic, could be confined to this category.

NO_2 interval-wise plotting

Likewise, for NO_2, the number of days in each year on which monitored values exceed the 24-hourly standards of 80 µg/m^3 is plotted (Figure 8-6).

First, we create a data frame for NO_2 that captures the number of days in each year with values higher than 80 µg/m^3, as shown in the following.

CODE:

```
#Calculating the number of days in each year with regard to critical days
of NO2 concentration
a=aq2015[(aq2015['NO2']>=80)]['NO2'].count()
b=aq2016[(aq2016['NO2']>=80)]['NO2'].count()
c=aq2017[(aq2017['NO2']>=80)]['NO2'].count()
d=aq2018[(aq2018['NO2']>=80)]['NO2'].count()
e=aq2019[(aq2019['NO2']>=80)]['NO2'].count()
dfno=pd.DataFrame({'years':['2015','2016','2017','2018','2019'],'number of
days with NO2>80 µg':[a,b,c,d,e]})
ax=dfno.plot(kind='bar',figsize=(10,5))
ax.set_xticklabels(['2015','2016','2017','2018','2019'])
ax.set_title("NO2 number of days in each year with critical levels of
concentration")
for p in ax.patches:
    ax.annotate(p.get_height(), (p.get_x() + p.get_width() / 2, p.get_
    height()), ha = 'center', va = 'bottom')
```

Output:

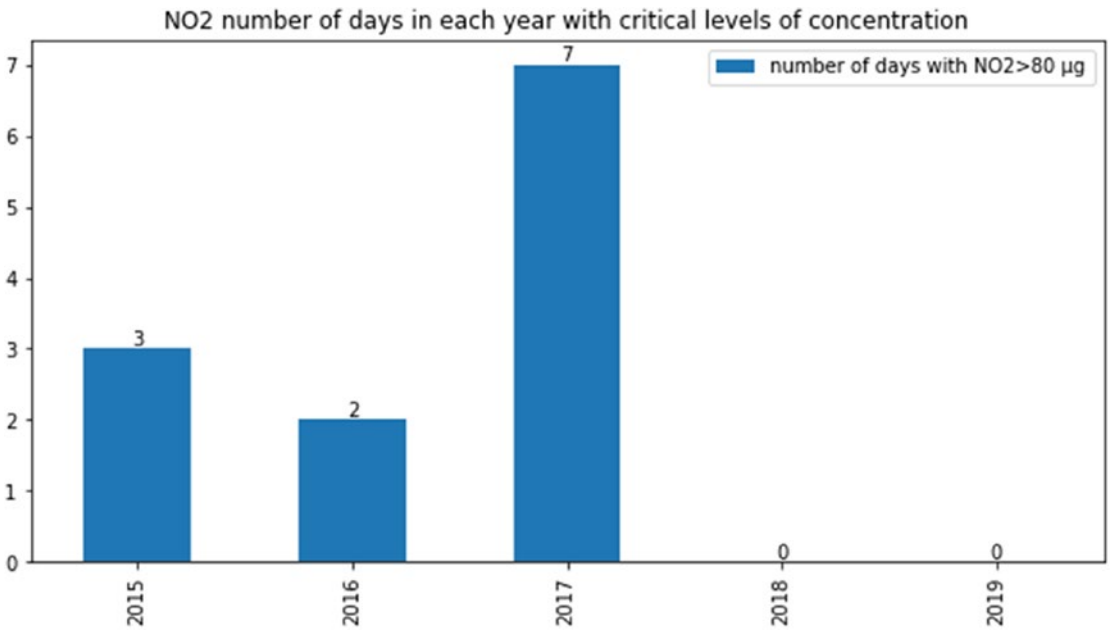

Figure 8-6. *Number of days per year with critical levels of NO_2 concentration*

Inference: Observed 24-hourly NO_2 values are exceeded only for three of the five years.

Since observed 24-hourly NO_2 values exceed standard only marginally and that too only for a few days, the next step is confined to further analysis of $PM_{2.5}$.

Part 3 of air quality analysis: Identifying the months where $PM_{2.5}$ daily values exceed critical levels on the majority of the days

Before imposing restrictions on activities like vehicular traffic and construction, which significantly contribute to ambient $PM_{2.5}$ concentrations, it is necessary to provide sufficient notice to avoid inconvenience to the general public. Hence, for daily $PM_{2.5}$ values significantly above 180 μg/m^3, we plot temporal variation year-wise during each month of the year. To do this, for each of the twelve months, we capture the number of critical air pollution days every year with 24-hourly $PM_{2.5}$ values exceeding 180 μg/m^3.

First, we create data frames for each year with the number of days in each month where the $PM_{2.5}$ values exceed 180 μg/m^3, as shown in the following.

CODE:

```
#Creating a dataframe for 2016 with the number of days in each month where
the PM2.5 concentration is >180
aq2016['Month']=pd.DatetimeIndex(aq2016['From Date']).month #extracting the
month
aq2016['condition']=(aq2016['PM2.5']>=180 ) # creating a boolean column
that is True when the PM2.5 value is greater than 180 and false when it is
less than 180
aq2016['condition']=aq2016['condition'].replace({False:np.nan}) # replacing
the False values with null values, so that the count method in the next
statement only counts the True values or the values corresponding to PM
2.5>180
selection1=aq2016.groupby('Month')['condition'].count() #Using the groupby
method to calculate the number of days for each month that satisfy the
condition(PM2.5>180)
#Repeating the above process for 2017, creating a dataframe with the number
of days in each month where the PM2.5 concentration is >180
aq2017['Month']=pd.DatetimeIndex(aq2017['From Date']).month
aq2017['condition']=(aq2017['PM2.5']>=180 )
aq2017['condition']=aq2017['condition'].replace({False:np.nan})
```

```
selection2=aq2017.groupby('Month')['condition'].count()
#Repeating the above process for 2018, creating a dataframe with the number
of days in each month where the PM2.5 concentration is >180
aq2018['Month']=pd.DatetimeIndex(aq2018['From Date']).month
aq2018['condition']=(aq2018['PM2.5']>=180 )
aq2018['condition']=aq2018['condition'].replace({False:np.nan})
selection3=aq2018.groupby('Month')['condition'].count()
#Repeating the above process for 2019, creating a dataframe with the number
of days in each month where the PM2.5 concentration is >180
aq2019['Month']=pd.DatetimeIndex(aq2019['From Date']).month
aq2019['condition']=(aq2019['PM2.5']>=180 )
aq2019['condition']=aq2019['condition'].replace({False:np.nan})
selection4=aq2019.groupby('Month')['condition'].count()
```

Now, we concatenate all the DataFrame objects into one object (which we will call 'selectionc') to get a consolidated picture of the number of days in each month where $PM_{2.5} > 180\ \mu g/m^3$, as shown in the following.

CODE:

```
#selectionc data frame is a consolidated dataframe showing month-wise
critical values of PM2.5 for every year
selectionc=pd.concat([selection1,selection1,selection3,selection4],axis=1)
#renaming the columns
selectionc.columns=['2016','2017','2018','2019']
selectionc
```

Output:

Month	2016	2017	2018	2019
1	20	20	23	14
2	3	3	5	3
3	1	1	0	0
4	3	3	0	1
5	3	3	0	2
6	7	7	4	1
7	2	2	0	0
8	0	0	0	0
9	2	2	0	0
10	5	5	5	4
11	13	13	7	11
12	4	4	11	18

We can observe from this table that month 1 (January), month 11 (November), and month 12 (December), are the most critical months for all four years, as these months had the highest number of days with $PM_{2.5} > 180$ μg/m^3.

Now that we have all the data in place, let us visualize the critical days for $PM_{2.5}$ (Figure 8-7), using the following code.

CODE:

```
#creating a bar chart representing number of days with critical levels of
PM2.5(>180) concentrations
ax=selectionc.plot(kind='bar',figsize=(20,7),width=0.7,align='center',color
map='Paired')
bars = ax.patches
#creating patterns to represent each year
patterns =('-','x','/','O')
#ax.legend(loc='upper left', borderpad=1.5, labelspacing=1.5)
ax.legend((patterns),('2016','2017','2018','2019'))
hatches = [p for p in patterns for i in range(len(selectionc))]
#setting a pattern for each bar
```

```
for bar, hatch in zip(bars, hatches):
    bar.set_hatch(hatch)
#Labelling the months, the X axis and Y axis
ax.set_xticklabels(['Jan','Feb','Mar','Apr','May','June','July','Aug','Sept
','Oct','Nov','Dec'],rotation=30)
ax.set_xlabel('Month',fontsize=12)
ax.set_ylabel('Number of days with critical levels of PM2.5',fontsize=12)
#Labelling the bars
for i in ax.patches:
    ax.text(i.get_x()-.003, i.get_height()+.3,
            round(i.get_height(),2), fontsize=10,
                color='dimgrey')
ax.legend()
ax.set_title("Number of days with critical levels of PM2.5 in each month of
years 2016-19")
```

Output:

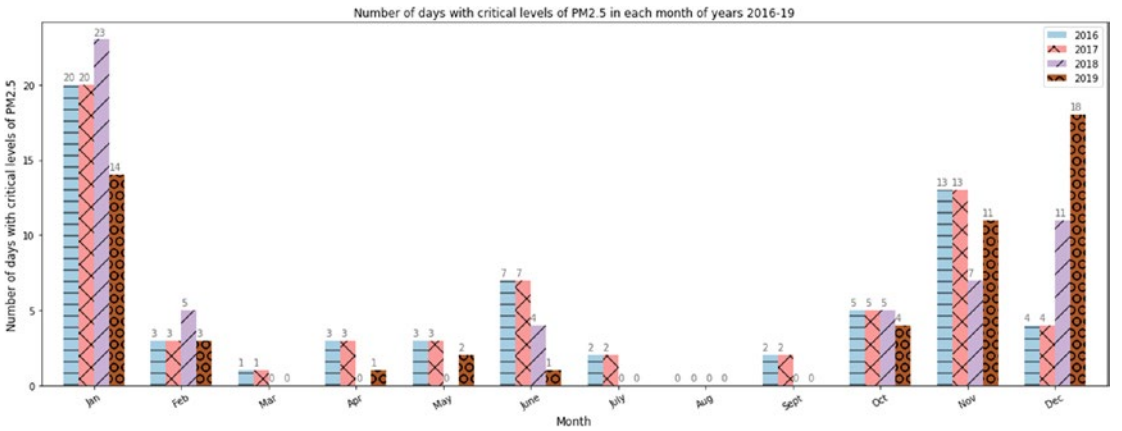

Figure 8-7. *$PM_{2.5}$ – Number of days with critical levels per month per year*

Step 4: Drawing inferences based on analysis and visualizations

From the preceding graph, it is observed that most of the critically polluted days fall in January, November, and December. Therefore, based on daily average concentrations of $PM_{2.5}$ recorded over the past four years, restrictions on vehicular traffic, construction activities, use of diesel pump sets, diversion of traffic entering Delhi from neighboring

states, and other similar activities are likely to be imposed in January, November, and December. To make such decisions for Delhi as a whole, analysis of data from other monitoring stations would also be necessary. Dissemination of data and analysis on the preceding lines would help people prepare in advance for restrictions and also appreciate the rationale behind such measures.

The approach demonstrated in the preceding, using data analysis as a tool to assist in air quality management, uses the data recorded at one monitoring station located at the Netaji Subhas Institute of Technology (NSIT), Delhi. The methodology could be carried forward on the following lines.

a. Repeat the preceding step for NO_2 to show critical months that account for most of the days with NO_2 recorded values exceeding 24-hourly standards. Doing this exercise would again help identify months facing pollution levels of concern for both parameters, $PM_{2.5}$ and NO_2, and plan.

b. Repeat the analysis carried out with data from the air quality monitoring station at NSIT with the use of similar data from other stations so that interventions for Delhi as a whole could be planned.

Case study 8-3: Worldwide COVID-19 cases – an analysis

This dataset contains data about the geographic distribution of COVID-19 cases as of 29th June 2020 (Source: European Center for Disease Control, source URL: https://www.ecdc.europa.eu/en/publications-data/download-todays-data-geographic-distribution-covid-19-cases-worldwide). Note that this link contains the latest data, but we have used the data as on 29th June (the link to the dataset is provided in the "Technical requirements" section at the beginning of the chapter).

Questions that we want to answer through our analysis include:

1. Which are the countries with the worst mortality rates, maximum cases, and the most fatalities?

2. What is the monthly trend vis-à-vis the number of cases and fatalities since the start of the pandemic?
3. In some of the countries, lockdowns were imposed to help flatten the curve. Did this measure aid in reducing the number of cases?

Step 1: Importing the data and examining the characteristics of the dataset

Read the dataset and examine the first five rows (using the *head* method) using the *pd.read_excel* function:

CODE:

```
df=pd.read_excel('COVID-19-geographic-distribution-worldwide-2020-06-29.xlsx')
df.head()
```

Output:

	dateRep	day	month	year	cases	deaths	countriesAndTerritories	geoId	countryterritoryCode	popData2019	continentExp
0	2020-06-29	29	6	2020	351	18	Afghanistan	AF	AFG	38041757.0	Asia
1	2020-06-28	28	6	2020	165	20	Afghanistan	AF	AFG	38041757.0	Asia
2	2020-06-27	27	6	2020	276	8	Afghanistan	AF	AFG	38041757.0	Asia
3	2020-06-26	26	6	2020	460	36	Afghanistan	AF	AFG	38041757.0	Asia
4	2020-06-25	25	6	2020	234	21	Afghanistan	AF	AFG	38041757.0	Asia

Get information about the data type of each column and the number of non-null values in each column (using the *info* method).

CODE:

```
df.info()
```

Output:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 26562 entries, 0 to 26561
Data columns (total 11 columns):
dateRep                    26562 non-null datetime64[ns]
day                        26562 non-null int64
month                      26562 non-null int64
year                       26562 non-null int64
cases                      26562 non-null int64
```

```
deaths                    26562 non-null int64
countriesAndTerritories   26562 non-null object
geoId                     26455 non-null object
countryterritoryCode      26498 non-null object
popData2019               26498 non-null float64
continentExp              26562 non-null object
dtypes: datetime64[ns](1), float64(1), int64(5), object(4)
memory usage: 2.2+ MB
```

Get summary statistics for each column (using the *describe* method) and obtain the values of the count, min, max, standard deviation, and percentiles:

CODE:

```
df.describe()
```

Output:

	day	month	year	cases	deaths	popData2019
count	26562.000000	26562.000000	26562.000000	26562.000000	26562.000000	2.649800e+04
mean	16.207929	4.194790	2019.997478	380.722611	18.882690	4.689196e+07
std	8.745421	1.555569	0.050161	2172.430663	121.386696	1.675462e+08
min	1.000000	1.000000	2019.000000	-2461.000000	-1918.000000	8.150000e+02
25%	9.000000	3.000000	2020.000000	0.000000	0.000000	1.919968e+06
50%	17.000000	4.000000	2020.000000	4.000000	0.000000	8.776119e+06
75%	24.000000	5.000000	2020.000000	68.000000	1.000000	3.194979e+07
max	31.000000	12.000000	2020.000000	54771.000000	4928.000000	1.433784e+09

Step 2: Data wrangling

In this step, we will:

- Check if the data types of the columns are accurately identified. If not, change the data types: From the output of the *info* method, we see that all data types of the columns have been correctly identified.
- Rename the columns if necessary: In the following code, we are renaming the columns of the DataFrame.

CODE:

```
#changing the column names
df.columns=['date','day','month','year','cases','deaths','country',
'old_country_code','country_code','population','continent']
```

- Drop any unnecessary columns or rows:
- We see the country code column is present twice (with two different names: 'old_country_code' and 'country_code') in the DataFrame, hence we remove one of the columns ("old_country_code"):

  CODE:

  ```
  #Dropping the redundant column name
  df.drop(['old_country_code'],axis=1,inplace=True)
  ```

- Remove any extraneous data that does not add any value:

  There are no blank spaces, special characters, or any other extraneous characters in this DataFrame. We see that there is data for only one day in December 2019; hence we remove data for this month and create a new DataFrame (df1) for the remaining 11 months.

  CODE:

  ```
  df1=df[df.month!=12]
  ```

- Check if there are any null values, using the *isna* or *isnull* method, and apply appropriate methods to deal with them if they are present:

  Calculating the percentage of null values:

  CODE:

  ```
  df1.isna().sum().sum()/len(df1)
  ```

  Output:

  ```
  0.008794112096622004
  ```

  Since the percentage of null values is less than 1%, we drop the null values in the following step.

CODE:

```
df1.dropna(inplace=True)
```

- Aggregate the data if the data is in a disaggregated format:

  The data in this DataFrame is not in an aggregated format, and we convert it into this format using the *groupby* method in this step. We can group either by country, by continent, or by date. Let us group by the name of the country.

  CODE:

```
#Aggregating the data by country name
df_by_country=df1.groupby('country')['cases','deaths'].sum()
df_by_country
```

  Output (only first nine rows shown):

country	cases	deaths
Afghanistan	30967	721
Albania	2402	55
Algeria	13273	897
Andorra	855	52
Angola	267	11
Anguilla	3	0
Antigua_and_Barbuda	65	3
Argentina	57731	1217
Armenia	25127	433

The preceding output shows a consolidated picture of the number of cases and deaths for each country.

Let us add another column to this aggregated DataFrame – the mortality rate, which is the ratio of the number of deaths to the number of cases.

CODE:

```
#Adding a new column for the mortality rate which is the ratio of the
number of deaths to cases
df_by_country['mortality_rate']=df_by_country['deaths']/df_by_
country['cases']
```

Step 3: Visualizing the data

In our first visualization in this case study, we use the aggregated data in the DataFrame, "df_by_country", to display the top twenty countries by mortality rate (Figure 8-8).

CODE:

```
#Sorting the values for the mortality rate in the descending order
plt.figure(figsize=(15,10))
ax=df_by_country['mortality_rate'].sort_values(ascending=False).head(20).
plot(kind='bar')
ax.set_xticklabels(ax.get_xticklabels(), rotation=45, ha="right")
for p in ax.patches:
    ax.annotate(p.get_height().round(2),(p.get_x()+p.get_width()/2,p.get_he
ight()),ha='center',va='bottom')
ax.set_xlabel("Country")
ax.set_ylabel("Mortality rate")
ax.set_title("Countries with highest mortality rates")
```

Output:

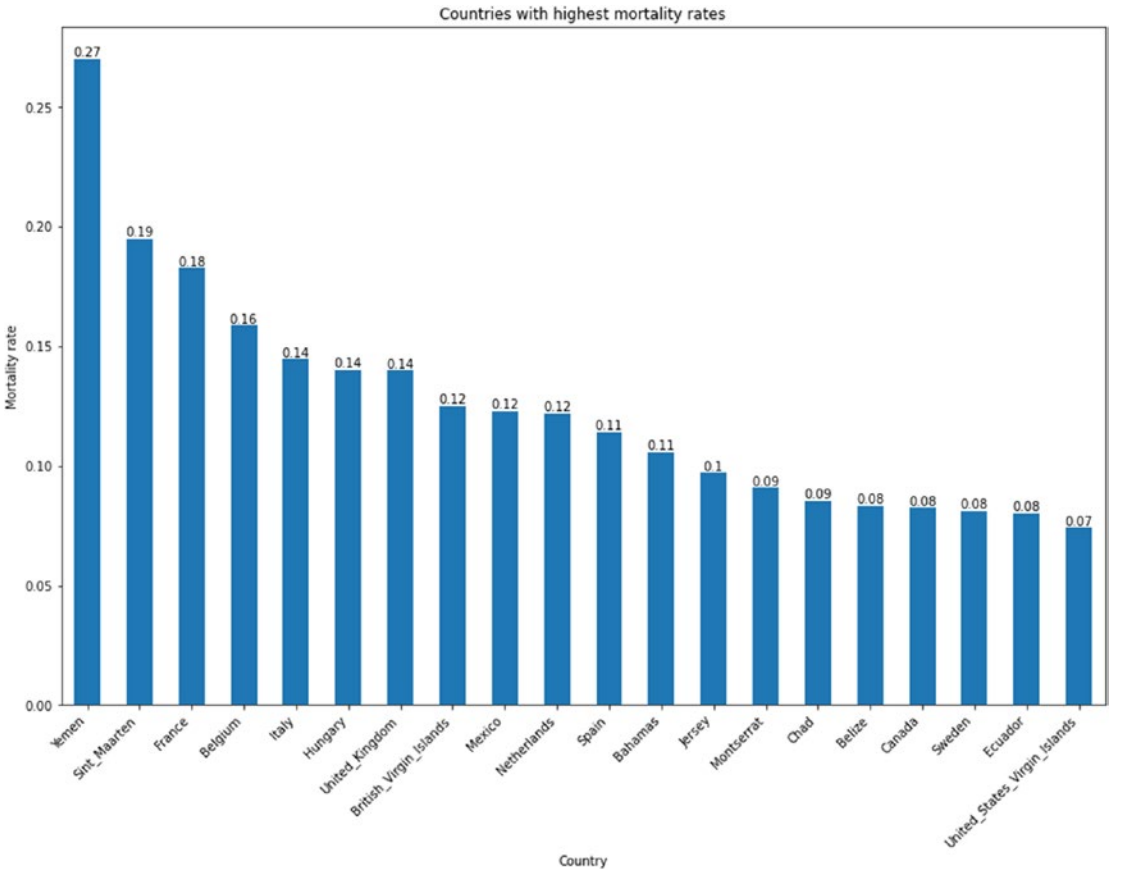

Figure 8-8. *Bar chart depicting countries with the highest mortality rates for COVID-19*

In the second visualization, we display the ten countries with the highest number of COVID-19 cases, using a pie chart, as shown in Figure 8-9.

CODE:

```
#Pie chart showing the countries with the highest number of COVID cases
df_cases=df_by_country['cases'].sort_values(ascending=False)
ax=df_cases.head(10).plot(kind='pie',autopct='%.2f%%',labels=df_cases.
index,figsize=(12,8))
ax.set_title("Top ten countries by case load")
```

Output:

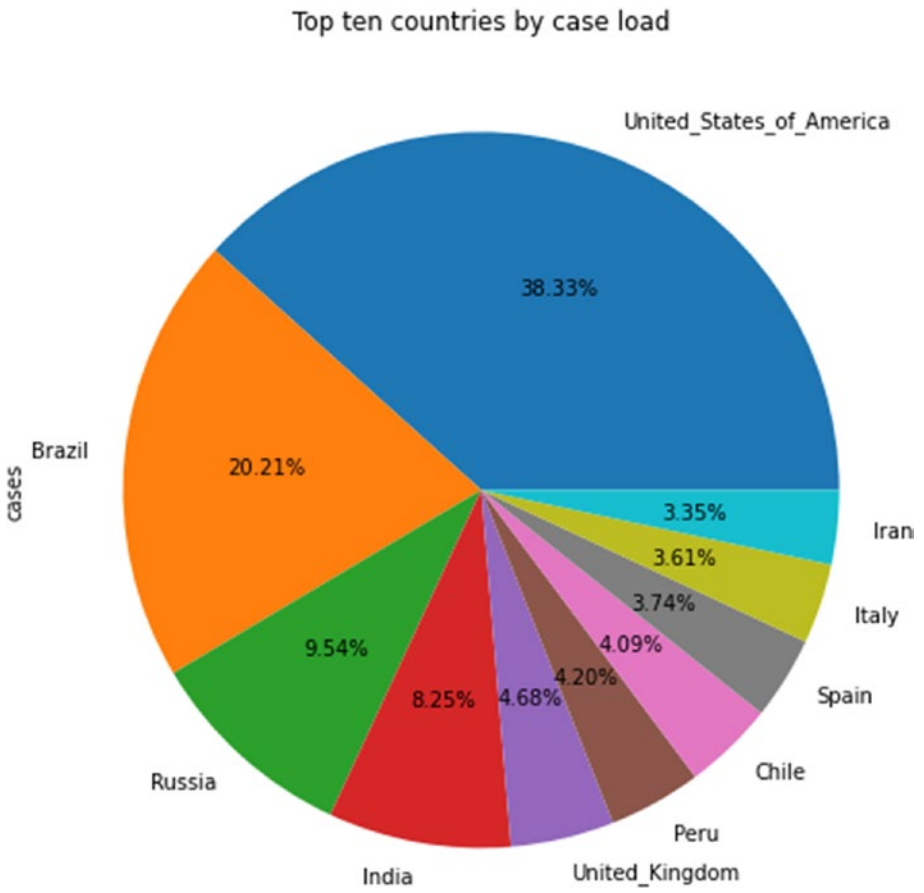

Figure 8-9. *Pie chart depicting the share of the top ten countries by COVID-19 cases*

In the next visualization, we find out the five countries that have suffered the most in terms of loss to human life from the COVID-19 pandemic, using a bar chart (Figure 8-10).

CODE:

```
#sorting the number of deaths in the descending order
plt.figure(figsize=(10,6))
ax=df_by_country['deaths'].sort_values(ascending=False).head(5).
plot(kind='bar')
ax.set_xticklabels(ax.get_xticklabels(), rotation=45, ha="right")
for p in ax.patches:
    ax.annotate(p.get_height(),(p.get_x()+p.get_width()/2,p.get_height()),
    ha='center',va='bottom')
```

```
ax.set_title("Countries suffering the most fatalities from COVID-19")
ax.set_xlabel("Countries")
ax.set_ylabel("Number of deaths")
```

Output:

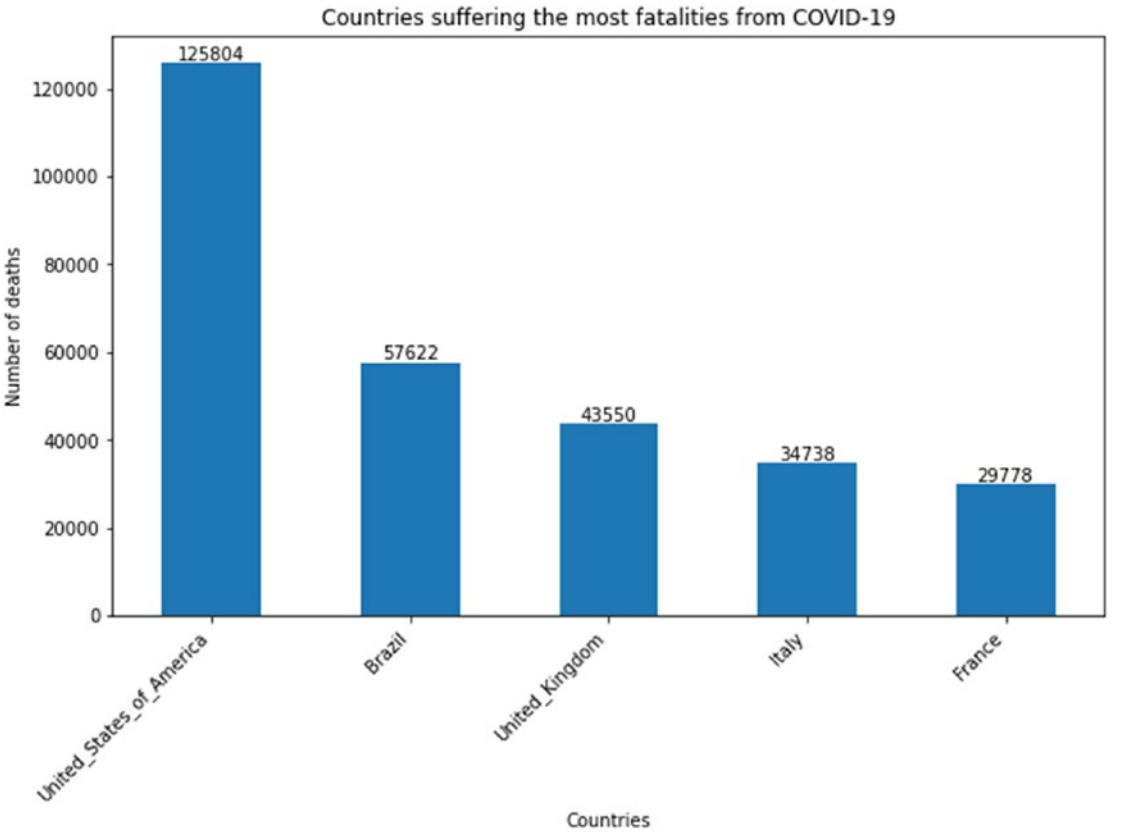

Figure 8-10. *Bar chart depicting the five countries with the maximum fatalities*

Now, we plot line graphs to see the month-wise trend in the number of COVID-19 cases and fatalities.

To plot the line graphs, we first aggregate the data by month and then plot two line graphs side by side, showing the number of cases and deaths, as shown in Figure 8-11.

CODE:

```
df_by_month=df1.groupby('month')['cases','deaths'].sum()
fig=plt.figure(figsize=(15,10))
ax1=fig.add_subplot(1,2,1)
```

```
ax2=fig.add_subplot(1,2,2)
df_by_month['cases'].plot(kind='line',ax=ax1)
ax1.set_title("Total COVID-19 cases across months in 2020")
ax1.set_xlabel("Months in 2020")
ax1.set_ylabel("Number of cases(in million)")
df_by_month['deaths'].plot(kind='line',ax=ax2)
ax2.set_title("Total COVID-19 deaths across months in 2020")
ax2.set_xlabel("Months in 2020")
ax2.set_ylabel("Number of deaths")
```

Output:

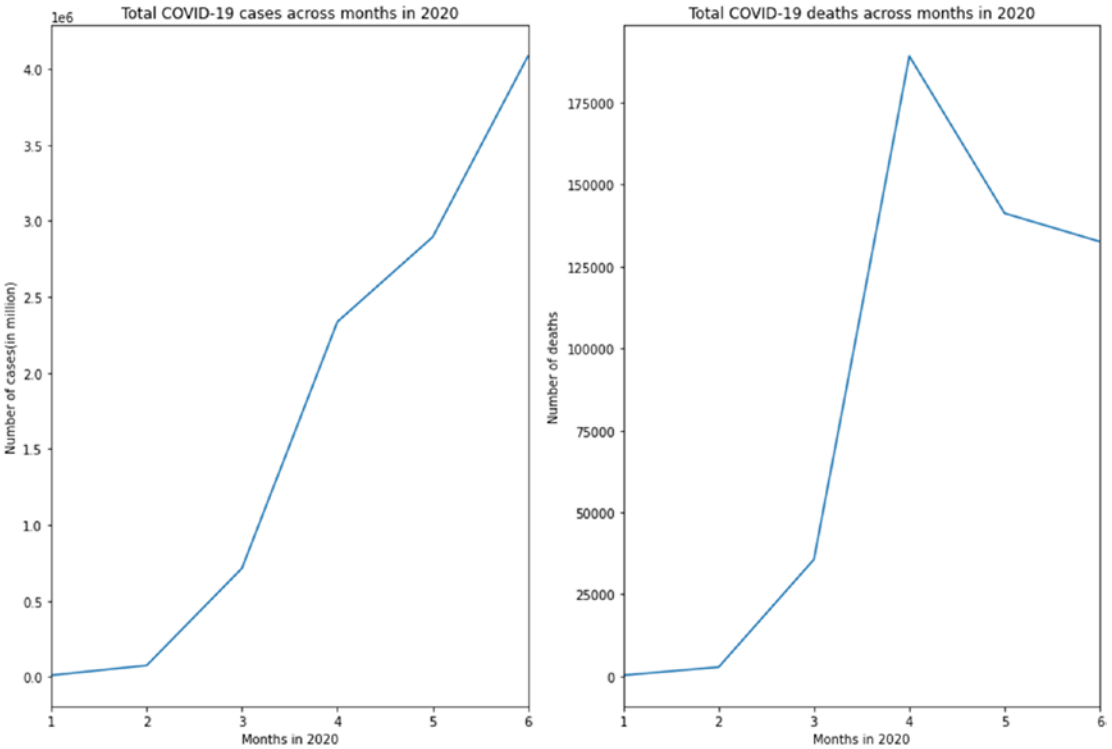

Figure 8-11. *Impact of lockdown on flattening the curve*

Many countries imposed a lockdown to stem the tide of increasing cases and flatten the curve. We now look at four countries – India, the UK, Italy, and Germany – where lockdowns were imposed in March, to see if this measure had the desired impact.

First, we create DataFrame objects for each of these countries, with data aggregated month-wise.

CODE:

```
#Creating DataFrames for each country
#Monthwise aggregated data for Germany
df_germany=df1[df1.country=='Germany']
df_germany_monthwise=df_germany.groupby('month')['cases','deaths'].sum()
df_germany_grouped=df_germany_monthwise.reset_index()
#Monthwise aggregated data for UK
df_uk=df1[df1.country=='United_Kingdom']
df_uk_monthwise=df_uk.groupby('month')['cases','deaths'].sum()
df_uk_grouped=df_uk_monthwise.reset_index()
#Monthwise aggregated data for India
df_india=df1[df1.country=='India']
df_india_monthwise=df_india.groupby('month')['cases','deaths'].sum()
df_india_grouped=df_india_monthwise.reset_index()
#Monthwise aggregated data for Italy
df_italy=df1[df1.country=='Italy']
df_italy_monthwise=df_italy.groupby('month')['cases','deaths'].sum()
df_italy_grouped=df_italy_monthwise.reset_index()
```

Now, we use the DataFrame objects created in the previous steps to plot line graphs for these countries to see the number of cases across various months in 2020, as shown in Figure 8-12.

CODE:

```
#Plotting the data for four countries (UK, India, Italy and Germany) where
lockdowns were imposed
fig=plt.figure(figsize=(20,15))
ax1=fig.add_subplot(2,2,1)
df_uk_grouped.plot(kind='line',x='month',y='cases',ax=ax1)
ax1.set_title("Cases in UK across months")
ax2=fig.add_subplot(2,2,2)
df_india_grouped.plot(kind='line',x='month',y='cases',ax=ax2)
ax2.set_title("Cases in India across months")
ax3=fig.add_subplot(2,2,3)
df_italy_grouped.plot(kind='line',x='month',y='cases',ax=ax3)
```

```
ax3.set_title("Cases in Italy across months")
ax4=fig.add_subplot(2,2,4)
df_germany_grouped.plot(kind='line',x='month',y='cases',ax=ax4)
ax4.set_title("Cases in Germany across months")
```

Output:

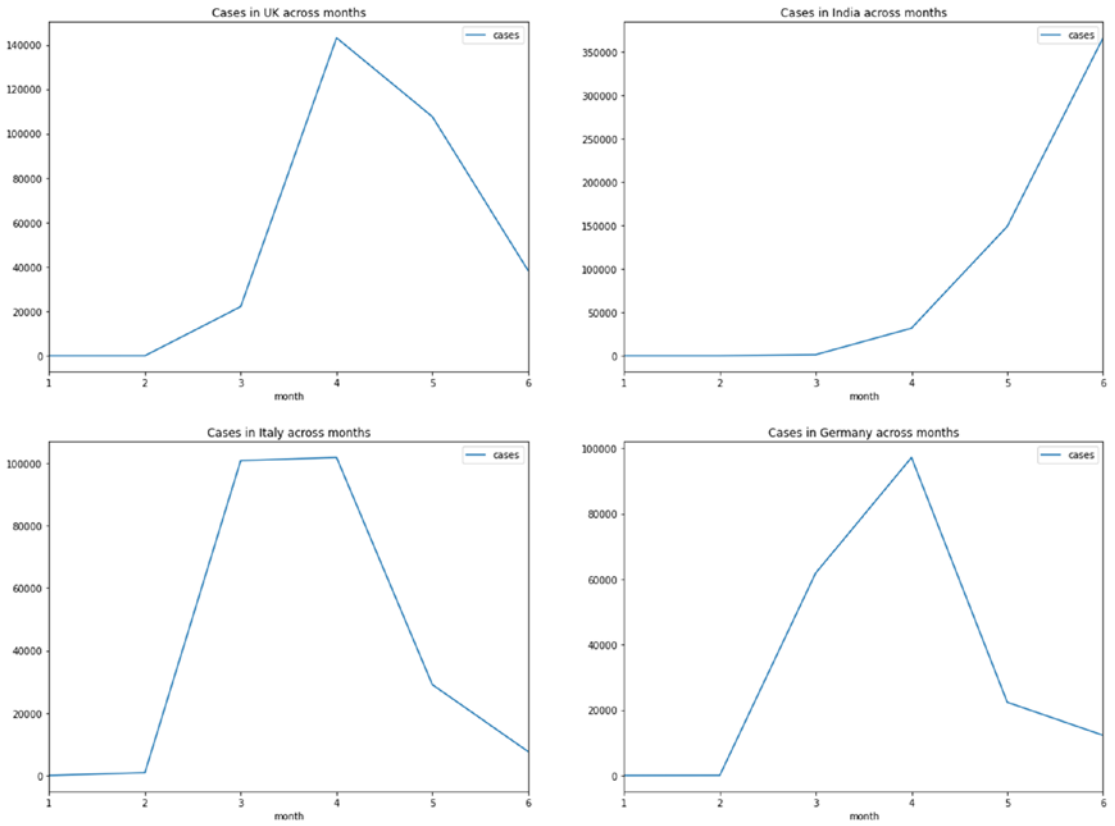

Figure 8-12. *Total cases in UK, India, Germany, and Italy in the first 6 months of 2020*

Step 4: Drawing inferences based on analysis and visualizations

- Number of cases: The United States, Brazil, Russia, India, and the UK had the highest number of cases.
- Number of deaths: The United States, Brazil, the UK, Italy, and France had the highest death tolls.

- Mortality rate: Yemen, St. Maarten, France, Belgium, and Italy had the highest mortality rates.
- Trends:
  - The total number of cases has been increasing steadily, while the total number of fatalities (deaths) has shown a decrease after April.
  - Impact of lockdown: We analyzed four countries – India, the UK, Germany, and Italy – where lockdowns were imposed in March. Except for India, all these countries experienced an overall decrease in cases after the lockdown was imposed. In the UK and Germany, the cases went up initially (during the early phase of the lockdown) but started decreasing after this spike.

Summary

- In this chapter, we looked at various case studies where we imported data from both structured and unstructured data sources. Pandas provides support for reading data from a wide variety of formats.
- The requests module has functions that enable us to send HTTP requests to web pages and store the content from the page in an object.
- A typical descriptive or exploratory data analysis of a case starts with framing the questions that we want to answer through our analysis and figuring out how to import the data. After this, we get more information about the data – the meanings of various columns, the units of measurement, the number of missing values, the data types of different columns, and so on.
- Data wrangling, where we prepare, clean, and structure the data to make it suitable for analysis, is the crux of descriptive or exploratory data analysis. Typical activities involved removing extraneous data, handling null values, renaming columns, aggregating data, and changing data types.

- Once the data is prepared and made suitable for analysis, we visualize our data using libraries like Matplotlib, Seaborn, and Pandas to help us gain insights that would answer the questions we initially framed.

Review Exercises

Question 1 (mini case study)

Consider the first table on the following web page: https://en.wikipedia.org/wiki/Climate_of_South_Africa. It contains data about the maximum and minimum temperatures (in degrees centigrade) in various cities in South Africa, during summers and winters.

- Use the appropriate method from the *requests* module to send a *get* request to this URL and store the data from the first table on this page in a Pandas DataFrame.
- Rename the columns as follows: 'City', 'Summer(max)', 'Summer(min)', 'Winter(max)', 'Winter(min)'.
- Replace the negative value in the first row of the 'Winter(min)' column with 0, and convert the data type of this column to *int64*.
- Plot a graph to display the hottest cities in South Africa during summers (use the Summer(max) column).
- Plot a graph to display the coldest cities in South Africa during the winters (use the Winter(min) column).

Question 2

The weekly wages of ten employees (with the initials A–J) are as follows: 100, 120, 80, 155, 222, 400, 199, 403, 345, 290. Store the weekly wages in a DataFrame.

- Plot a bar graph to display the wages in the descending order
- Label each of the bars in the bar graphs using the *annotate* method

Question 3

1. Module for sending HTTP requests	a. URL of a web page
2. *Get* method	b. req.text
3. Argument passed to *get* method	c. Fetches information using a given URL
4. Attribute containing Unicode content	d. Requests

Question 4

The *read_html* Pandas function reads

1. All the HTML content on the web page
2. HTML tags in a web page
3. All the HTML tables as a list of DataFrame objects
4. HTML lists in a web page

Answers

Question 1

CODE:

```
import requests
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline
url='https://en.wikipedia.org/wiki/Climate_of_South_Africa'
#making a get request to the URL
req = requests.get(url)
#storing the HTML data in a DataFrame
data = pd.read_html(req.text)
#Reading the first table
df=data[0]
#Renaming the columns
df.columns=['City','Summer(max)','Summer(min)','Winter(max)','Winter(min)']
#Replacing the negative value with 0
df['Winter(min)']=df['Winter(min)'].str.replace(r"−2","0")
```

```
#Changing the data type from object to int64
df['Winter(min)']=df['Winter(min)'].astype('int64',errors='ignore')
#Using the city as the index to facilitate plotting
df1=df.set_index('City')
#Hottest five cities during Summer
df1['Summer(max)'].sort_values(ascending=False).head(5).plot(kind='bar')
#Coldest five cities during Winter
df1['Winter(min)'].sort_values().head(5).plot(kind='bar')
```

Question 2

CODE:

```
numbers=pd.Series([100,120,80,155,222,400,199,403,345,290])
#converting the data to a DataFrame
numbers.to_frame()
#labelling the index
numbers.index=list('ABCDEFGHIJ')
#labelling the column
numbers.columns=['Wages']
ax=numbers.sort_values(ascending=False).plot(kind='bar')
#labelling the bars
for p in ax.patches:
    ax.annotate(p.get_height(),(p.get_x()+p.get_width()/2,p.get_height()),h
a='center',va='bottom')
```

Question 3

1-d; 2-c; 3-a; 4-b

Question 4

Option 3

Content from each table on the web page is stored in a separate DataFrame object.